Pointers

In Fortran, a *pointer variable* or simply a *pointer* is best thought of as a ``free-floating" name that may be associated dynamically with or ``aliased to" some data object. The data object already may have one or more other names or it may be an unnamed object.

Syntactically, a pointer is just any sort of variable that has been given the pointer attribute in a declaration. A variable with the pointer attribute may be used just like any ordinary variable, but it may be used in some additional ways as well.

Next slide

The States of a Pointer

Each pointer in a program is in one of the three following states:

1. It may be *undefined*, which is the condition of all pointers at the beginning of a program.
2. It may be *null*, which means that it is not the alias of any data object.
3. It may be *associated*, which means that it is the alias of some target data object.

The term ``disassociated" is used when a pointer is in state 2. Thus, the `associated` intrinsic inquiry function distinguishes between states 2 and 3 only.

Learn more about pointers.

Previous slide Next slide

The Pointer Assignment Statement

```
real, pointer :: p
real, target :: r

p => r
```

This statement causes `p` to point to `r`, or causes `p` to be an alias for `r`.

Previous slide Next slide

The `target` Attribute

Any variable aliased or ``pointed to'' by a pointer must be given the *target attribute* when declared and it must have the same type, kind, and rank as the pointer.

However, it is not necessary that the variable have a defined value.

Learn more about the `pointer` attibute.
Learn more about the `target` attibute.

Previous slide Next slide

A variable with the pointer attribute may be an object more complicated than a simple variable. It may be an array section or structure, for example. The following declares `v` to be a pointer to a one-dimensional array of reals:

```
real, dimension (:), pointer :: v
real, dimension (40, 60), target :: real_array
```

With `v` so declared, it may be used to alias any one-dimensional array of reals, including a row or column of `real_array`.

```
v => real_array (4, :)
```

Previous slide Next slide

Use of Pointer Variables

Once a variable with the pointer attribute is an alias for some data object, that is, it is pointing to something, it may be used in the same way that any other variable may be used. For the example above using `v`,

```
print *, v
```

has exactly the same effect as

```
print *, real_array (4, :)
```

and the assignment statement

```
v = 0
```

has the effect of setting all of the elements of the fourth row of the array `real_array` to 0.

Previous slide Next slide

A different version of the pointer assignment statement occurs when the right side also is a pointer. This is illustrated by the following example, in which `p1` and `p2` are both real variables with the pointer attribute and `r` is a real variable with the target attribute.

```
real, target :: r
real, pointer :: p1, p2
r = 4.7
p1 => r
p2 => p1
r = 7.4
print *, p2
```

Previous slide Next slide

After execution of the first assignment statement

```
r = 4.7
```

`r` is a name that refers to the value 4.7:

```
           r
          ______
         |      |
         | 4.7  |
         |______|
```

Previous slide Next slide

The first pointer assignment

```
p1 => r
```

causes `p1` to be an alias for `r`, so that the value of the variable `p1` is 4.7. The value 4.7 now has two names, `r` and `p1`, by which it may be referenced.

```
          r
           _____
  p1 --> |     |
         | 4.7 |
         |_____|
```

Previous slide Next slide

The next pointer assignment

```
p2 => p1
```

causes `p2` to be an alias for the same thing that `p1` is an alias for, so the value of the variable `p2` is also 4.7. The value 4.7 now has three names or aliases, `r`, `p1`, and `p2`.

```
           r
          _____
  p1 --> |     |
  p2 --> | 4.7 |
         |_____|
```

Changing the value of `r` to 7.4 causes the value of both `p1` and `p2` also to change to 7.4 because they are both aliases of `r`. Thus, the next print statement

```
print *, p2
```

prints the value 7.4.

Previous slide Next slide

The pointer assignment statement

```
p => q
```

is legal whatever the status of `q`. If `q` is undefined, `p` is undefined; if it is null, `p` is nullified; and if it is aliased to or associated with a target, p becomes associated with the same target. Note that if `q` is associated with some target, say `t`, it is not necessary that `t` have a defined value.

Previous slide Next slide

The Difference between Pointer and Ordinary Assignment

Pointer assignment transfers the status of one pointer to another. In an ordinary assignment in which pointers occur, the pointers must be viewed simply as aliases for their targets.

```
real, pointer :: p1, p2
real, target  :: r1, r2
   .  .  .
r1 = 1.1;  r2 = 2.2
p1 => r1;  p2 => r2
```

This produces the following situation:

```
          r1                r2
          _____             _____
  p1 --> |     |   p2 --> |     |
         | 1.1 |          | 2.2 |
         |_____|          |_____|
```

Previous slide Next slide

Now suppose the ordinary assignment statement

```
p2 = p1
```

is executed. This statement has exactly the same effect as the statement

```
r2 = r1
```

because `p2` is an alias for `r2` and `p1` is an alias for `r1`. The situation is now:

```
          r1                r2
          _____             _____
  p1 --> |     |   p2 --> |     |
         | 1.1 |          | 1.1 |
         |_____|          |_____|
```

because the value 1.1 has been copied from `r1` to `r2`. The values of `p1`, `p2`, `r1`, and `r2` are all 1.1. Subsequent changes to `r1` or `p1` will have no effect on the value of `r2`.

Previous slide Next slide

If, on the other hand, the pointer assignment statement

```
p2 => p1
```

were executed instead, this statement would produce the situation

```
          r1                r2
          _____             _____
  p1 --> |     |           |     |
  p2 --> | 1.1 |           | 2.2 |
         |_____|           |_____|
```

In this case, too, the values of `p1`, `p2`, and `r1` are 1.1, but the value of `r2` remains 2.2. Subsequent changes to `p1` or `r1` do change the value of `p2`. They do not change the value of `r2`.

Previous slide Next slide

If the target of `p1` is changed to `r2` by the pointer assignment statement

```
p1 => r2
```

the target `r1` and value 1.1 of `p2` do not change, producing the following situation:

```
          r1                r2
          _____             _____
         |     |   p1 --> |     |
 p2 --> | 1.1 |           | 2.2 |
         |_____|           |_____|
```

The pointer `p2` remains an alias for `r1`; it does not remain associated with `p1`.

Previous slide Next slide

The `allocate` and `deallocate` Statements

With the `allocate` statement, it is possible to create space for a value and cause a pointer variable to refer to that space. The space has no name other than the pointer mentioned in the `allocate` statement.

```
allocate (p1)
```

creates space for one real number and makes `p1` an alias for that space. No real value is stored in the space by the `allocate` statement, so it is necessary to assign a value to `p1` before it can be used (unless it has been default initialized), just as with any other real variable.

```
          _____
p1 --> |       |
       |       |
       |_______|
```

Learn more about pointer allocation.

Previous slide Next slide

The statement

```
p1 = 7.7
```

sets up the following situation.

```
 p1 --> |‾‾‾‾‾‾|
        | 7.7  |
        |______|
```

Before a value is assigned to `p1`, it must either be associated with an unnamed target by an `allocate` statement or be aliased with a target by a pointer assignment statement.

Previous slide Next slide

The `deallocate` statement throws away the space pointed to by its argument and makes it null (state 2). For example,

```
deallocate (p1)
```

disassociates `p1` from any target and nullifies it.

```
  p1 -->
```

After `p1` is deallocated, it must not be referenced in any situation that requires a value; however it may be used, for example, on the right side of a pointer assignment statement. If other pointer variables were aliases for `p1`, they, too, no longer reference a value.

Previous slide Next slide

Pointers vs. Allocatable Arrays

Everything that can be done with allocatable arrays also can be done with pointers.

```
real, dimension (:, :), pointer :: matrix
   . . .
read *, n
allocate (matrix (n, n))
matrix = 0
   . . .
```

Previous slide Next slide

Exercise

1. Read a value for `n`, allocate a pointer array of `n` elements, put `n` random numbers into the array, sort the array, and print the first 10 and last 10 elements of the sorted array.

Previous slide Next slide

The `nullify` Statement and `null` Intrinsic Function

When a pointer is nullified, it may be tested and assigned to other pointers by pointer assignment (`=>`). A pointer is nullified with the `nullify` statement or `null` function.

```
nullify (p1)
pqr => null()
```

If the target of `p1` and `p2` are the same, nullifying `p1` does not nullify `p2`. On the other hand, if `p1` is null, then executing the pointer assignment

```
p2 => p1
```

causes `p2` to be null also.

A null pointer is not associated with any target or other pointer.

Learn more about pointer nullification.

Previous slide Next slide

The `associated` Intrinsic Function

The `associated` intrinsic function may be used to determine if a pointer variable is pointing to, or is an alias for, another object. To use this function, the pointer variable must be defined; that is, it must either be the alias of some data object or be null. The `associated` function indicates which of these two cases is true and so also provides the means of testing if a pointer is null.

The `associated` function may have a second argument. If the second argument is a target, the value of the function indicates whether the first argument is an alias of the second argument. If the second argument is a pointer, it must be defined; in this case, the value of the function is true if both pointers are null or if they are both aliases of the same target.

Previous slide Next slide

For example, the expression

```
associated (p1, r)
```

indicates whether or not `p1` is an alias of `r`, and the expression

```
associated (p1, p2)
```

indicates whether `p1` and `p2` are both aliases of the same thing or they are both null.

Learn more about pointer association.

Previous slide Next slide

Heat Equation Using Pointers

```
!  A simple solution to the heat equation using arrays
!  and pointers

program heat

real, dimension(10,10), target :: plate
real, dimension(8,8)           :: temp
real, pointer, dimension(:,:)  :: n, e, s, w, inside

real    :: diff
integer :: i,j, niter
```

Previous slide Next slide

```
! Set up initial conditions
plate = 0
plate(1:10,1) = 1.0  ! boundary values
plate(1,1:10) = (/ ( 0.1*j, j = 10, 1, -1 ) /)

!  Point to parts of the plate
inside => plate(2:9,2:9)
n => plate(1:8,2:9)
s => plate(3:10,2:9)
e => plate(2:9,1:8)
w => plate(2:9,3:10)
```

Previous slide Next slide

```
! Iterate
niter = 0
do
  temp = (n + e + s + w)/4.0
  diff = maxval(abs(temp-inside))
  niter = niter + 1
  inside = temp
  print *, niter, diff
  if (diff < 1.0e-4) then
    exit
  endif
end do

do i = 1,10
  print "(10f7.3)", plate(1:10,i)
enddo

end program heat
```

Previous slide Next slide

Tree Sort

```
program tree_sort
! Sorts a file of integers by building a
! tree, sorted in infix order.
! This sort has expected behavior n log n,
! but worst case (input is sorted) n ** 2.

   implicit none
   type node
      integer :: value
      type (node), pointer :: left => null(), &
                              right => null()
   end type node

   type (node), pointer :: tree_top => null()  ! A tree
   integer :: number, ios
```

Previous slide Next slide

```
   ! Start with empty tree
   do
      read (*, *, iostat = ios) number
      if (ios < 0) exit
      ! Put next number in tree
      call insert (tree_top)
   end do
   ! Print nodes of tree in infix order
   call print_tree (tree_top)

contains
```

Previous slide Next slide

```
   recursive subroutine insert (t)

      type (node), pointer :: t  ! A tree

      ! If (sub)tree is empty,
      ! put number at root
      if (.not. associated (t)) then
         allocate (t)
         t % value = number  ! Subtrees are null
      ! Otherwise, insert into correct subtree
      else if (number < t % value) then
         call insert (t % left)
      else
         call insert (t % right)
      end if

   end subroutine insert
```

Previous slide Next slide

```
   recursive subroutine print_tree (t)
   ! Print tree in infix order

      type (node), pointer :: t  ! A tree

      if (associated (t)) then
         call print_tree (t % left)
         print *, t % value
         call print_tree (t % right)
      end if

   end subroutine print_tree

end program tree_sort
```

Previous slide Next slide

Exercise

1. Implement a new data type `stack_of_integers`, with operations (functions) `new_empty_stack`, `is_stack_empty`, `top_of_stack`, `rest_of_stack` (that returns the stack without the top element), and `sorted (stack)` (that returns a stack consisting of the same elements of `stack`, but sorted with the smallest element at the top of the stack).

Previous slide